Bujar Sefa
CSC 21200 -BC
Professor Liu
HW #4

Overview of homework:

This homework is about creating the underlying data structures of stacks and queues, using dynamic arrays. I had to create the header file which contained the prototypes of the function each class needed; stack functions include but are not limited to pop, push, and size. Also, I had to create the implementation files of each function, i.e., for the size function, return the "space" (number of array spots) that was used. In addition, I had to create a priority queue data structure using dynamic arrays. Lastly, I had to implement a stack and queue using two of the opposite data structures as the private member variables, and only their functions to implement the basic functions for that stack or queue.

Stack Data Structure: <NOTE only stack is explained in this much detail.>

Header File: Sefa_Bujar_HW4_Q1.h:

```
#ifndef __STACK_H__
#define __STACK_H__

#include <iostream>
#include <cassert>
#include <cstdlib>
using namespace std;


template<class Item>
class stack{
    public:
        stack(int init_capacity = 30){
               capacity = init_capacity;
               used = 0;
               data = new Item[capacity];
        }

        ~stack();
        void print() const;
        void push(const Item& entry);
        size_t size() const;
        bool isEmpty() const;
        void pop();
        stack(const stack& source);
        void operator=(const stack& source);
        Item top() const;
```

```
   private:
        Item * data;
        size_t used;
        size_t capacity;

};

template<class Item>
void swapStacks(stack<Item>& stack1, stack<Item>& stack2);

#include "Sefa_B_HW4_Q1.cpp"
#endif
```

Implementation File: Sefa_Bujar_HW4_Q1.cpp:

stack():

What: This function/constructor creates a stack.

How: This function/constructor initialize all the member variables of the stack, creates a dynamic array for data, sets initial capacity, and used = 0.

Precondition: N/A.

Postcondition: A stack is created.

Worst Time Complexity: O(1)

/*NOTE written in header file*/

```
stack(int init_capacity = 30){
   capacity = init_capacity;
   used = 0;
   data = new Item[capacity];
}
```

~stack():

What: This function deletes or frees up space allocated by the stack.

How: This function utilizes C++ function to remove elements and free up space via the delete operator.

Precondition: N/A

Postcondition: The memory that this stack allocated is now freed, and all other variables are reset to 0.

Worst Time Complexity: O(n) because data is an array.

```
template<class Item>
stack<Item>::~stack(){
    delete [] data;
    used = 0;
    capacity = 0;
}
```

push()

What: This function inserts a new item into the top of the stack.

How: This function first checks to see if the stack has enough space to insert more items. If it does not, it allocates a larger stack and transfers all its current items into the new stack (Maintaining the order); it deletes the old stack to free up space. Then, it adds the new item to the end of the array (or top of the stack), and it increases the variable that keeps track of its size by 1.

Precondition: entry must be of type item.

Postcondition: The stack now has a new element, entry, at the top.

Worst Time Complexity: O(n) because of larger array allocation.

```
template<class Item>
void stack<Item>::push(const Item& entry){
    /*Check if we have enough space*/
    if(used>=size()){
        Item * larger = new Item[++capacity];
        for(size_t i = 0; i < used; i++){
               larger[i] = data[i];
        }
        delete [] data;
        data = larger;
    }
    data[used++]=entry;
}
```

isEmpty():

What: This function checks to see if the stack is empty.

How: This function compares the number of items in the array to 0, and returns true if so, or false if not.

Precondition: N/A

Postcondition: Returns if the stack has no items in it or not.

Worst Time Complexity: O(1)

```
template<class Item>
bool stack<Item>::isEmpty() const{
    return used == 0;
}
```

size():

What: This function returns the size of the stack.

How: This function returns the variable used, which is the variable that keeps track of the number of items in the stack.

Precondition: N/A

Postcondition: The size of the stack is returned.

Worst Time Complexity: O(1)

```
template<class Item>
size_t stack<Item>::size() const{
    return used; /*Just return the last one in the list.*/
}
```

pop():

What: This function removes the last item from the stack.

How: This function first checks to see if the stack is empty, and halts the program if it is true. Then, it just decreases the value of used, as then the stack wouldn't be able to see the previous last element (the invariant of the class).

Precondition: The stack must not be empty.

Postcondition: The top element of the stack is now removed.

Worst Time Complexity: O(1)

```
template<class Item>
void stack<Item>::pop(){
    assert(!isEmpty());
    used--;
}
```

stack(stack):

What: This function creates a new stack that is the copy of another.

How: This function copies over all the data from source stack into the current stack.

Precondition: N/A

Postcondition: A new stack is created that is a copy of the source stack.

Worst Time Complexity: O(n)

```
template<class Item>
stack<Item>::stack(const stack<Item>& source){
    /*NOTE no need to check if they are the same because stacks are not the same*/
    capacity = source.capacity;
    used = source.used;
    data = new Item[capacity];
    for(size_t i = 0; i < used; i++){
        data[i] = source.data[i];
    }
}
```

Operator =():

What: This function deletes or frees up space allocated by the stack.

How: This function first checks to see if the stack in the argument is nt the same as the one calling the function, if so do nothing because the stacks are the same. If not the case, then free up the space from the current stack, and copy over all elements from source into current stack (including used and capacity).

Precondition: N/A
Postcondition: The current stack is now a copy of source stack.
Worst Time Complexity: O(n).

```
template<class Item>
void stack<Item>::operator =(const stack<Item>& source){
    /*Check to see if source is the same thing, like y = y*/
    if(this == &source){ /*This is saying, if the address of current pointer, is same as the
address of the argument. */
        return; /*Dont do anything if both of their addresses are the same.*/
    }
    used = source.used;
    capacity = source.capacity;
    delete [] data; /*Free up any allocated space. */
    data = new Item[capacity];
    for(size_t i = 0; i < used; i++){
        data[i] = source.data[i];
    }
}
```

top():
What: This function returns the top item in the stack.
How: This funtion first checks to see if the stack is empty or not. If the case is that its empty, it halts the program. If not, then it returns the item that is at the top of the stack.
Precondition: Stack must contain items.
Postcondition: The top item of the stack is returned.
Worst Time Complexity: O(1)

```
template<class Item>
Item stack<Item>::top() const{
    assert(!isEmpty());
    return data[used-1]; /*Just return the last one in the list.*/
}
```

Explanation of Stack Swapping

swapStacks():
What: This function swaps the content of two stacks.
How: This function first checks to see if the two stacks are the same. If the case is true nothing happens. If it is not the case, then a temporary stack of **'this'** is created using the copy constructor. Then the temporary stack gets all of the elements of **this** stack. Following, all of information of the source stack are transferred into this stack. Finally, source stack is assigned the temporary stack.
Precondition: N/A
Postcondition: The two stacks this and source are swapped.
Worst Time Complexity: O(n)

```
template<class Item>
void stack<Item>::swapStacks(stack<Item>& source){
    if(this == &source)/*If they are the same stacks, dont do anything*/
        return;
    stack<Item> tempThis(capacity);
    tempThis.used = used;
    for(size_t i = 0; i < used; i++){
        tempThis.data[i] = data[i];
    }
    delete [] data;
    used = source.used;
    capacity = source.capacity;
    data = new Item[capacity];
    for(size_t i = 0; i < used; i++){
        data[i] = source.data[i];
    }
    source = tempThis;
}
```

Helper Method:

print():

What: This function prints out the items in the array (Not in stack order). Used only for testing.

How: This function loops through each element and prints out the item.

Precondition: N/A

Postcondition: Each item of the dynamic array is printed.

Worst Time Complexity: O(n)

```
template<class Item>
void stack<Item>::print() const{
    for(size_t i = 0; i < used; i++){
        cout << data[i] << endl;
    }
}
```

Queue Data Structure:

Header File: Sefa_Bujar_HW4_Q2.h:

```
#ifndef __QUEUE_H__
#define __QUEUE_H__

#include <iostream>
#include <cassert>
#include <cstdlib>
using namespace std;

template<class Item>
class queue{
    public:
        queue(size_t init_cap= 5); /*NOTE decreased capacity to smaller testable value. change
back to 30*/
        ~queue();
        queue(const queue<Item>& source);
        void operator=(const queue<Item>& source);
        void pop();
        void push(const Item& entry);
        Item front() const;
        size_t size() const; /*{ return count; }*/
        bool empty() const;
        /* TO IMPLEMENT LATER */
        void swapQueues(queue<Item>& source);
        void print();

    private:
        Item * data;
        size_t first;
        size_t last;
        size_t count;
        size_t capacity;
        /* Helper Method */
        size_t next_index(size_t i){ return (i+1)%capacity;}

};
```

Implementation File: Sefa_Bujar_HW4_Q2.cpp:

```
#include "Sefa_B_HW4_Q2.cpp"
#endif

#ifndef __QUEUE_CPP__
#define __QUEUE_CPP__

#include "Sefa_B_HW4_Q2.h"

template<class Item>
queue<Item>::queue(size_t init_cap){
    capacity = init_cap;
    data = new Item[capacity];
    count = 0;
    first = 0;
    last = capacity-1;/*Because next index gets the first index when want to start*/
}

template<class Item>
queue<Item>::~queue(){
    delete [] data;
    capacity = 0;
    first = 0;
    last = -1;
    count = 0;
}

template<class Item>
queue<Item>::queue(const queue<Item>& source){
    capacity = source.capacity;
    count = source.count;
    first = source.first;
    last = source.last;
    data = new Item[capacity];
    for(size_t i = 0; i < capacity; i++){
        data[i] = source.data[i];
    }
}

template<class Item>
void queue<Item>::operator=(const queue<Item>& source){
    if(this == &source)
        return;
    delete [] data;
    capacity = source.capacity;
    data = new Item[capacity];
    first= source.first;
    last = source.last;
    count = source.count;
    size_t i = first;
    for(size_t i =0; i < capacity; i++){
        data[i] = source.data[i];
    }
```

```
}

template<class Item>
bool queue<Item>::empty() const{
    return count == 0;
}

template<class Item>
Item queue<Item>::front() const {
    assert(!empty());
    return data[first];
}

template<class Item>
void queue<Item>::push(const Item& entry){
    if(count >= capacity){
        Item * bigger = new Item[++capacity];
        for(size_t i = 0; i < count; i++){
               bigger[i] = data[i];
        }
        delete [] data;
        this->data = bigger;
    }
    last = next_index(last);
    data[last] = entry;
    count++;

}

template<class Item>
void queue<Item>::pop(){
/*Check if queue is empty*/
    assert(!empty());
    /*THIS IS FIFO, thus, must get rid of first, NOT LAST*/
    first = next_index(first);
    count--;
}

template<class Item>
size_t queue<Item>::size() const{
    return count;
}

template<class Item>
void queue<Item>::print(){ /*NOTE just a regular print for testing.*/
    size_t index = 0;
    for(size_t i = 0; i < size(); i++){
        cout << data[i] << endl;
    }
}

template<class Item>
void queue<Item>::swapQueues(queue<Item>& source){
    if(this == &source)/*If they are the same stacks, dont do anything*/
```

```
        return;
    queue<Item> tempThis(capacity);
    for(size_t i = 0; i < capacity; i++){
        tempThis.data[i] = data[i];
    }
    tempThis.first = first;
    tempThis.last = last;
    tempThis.count = count;
    delete [] data;
    capacity = source.capacity;
    data = new Item[capacity];
    first = source.first;
    last = source.last;
    count = source.count;
    for(size_t i = 0; i < capacity; i++){
        data[i] = source.data[i];
    }
    source = tempThis;
}
#endif
```

Explanation of Swapping Queue:

What: This function swaps two queues.

How: This function first checks to see if the two queues are the same. If the case is true nothing happens. If it is not the case, then a temporary queue of **'this'** is created using the copy constructor. Then the temporary queues gets all of the elements of **this** queues. Following, all of information of the source queue are transferred into this queue. Finally, source queues is assigned the temporary queue.

Precondition: N/A

Postcondition: The two queues this and source are swapped.

Worst Time Complexity: O(n)

Priority Queue Data Structure:

Header File: Sefa_Bujar_HW4_Q3.h:

```
#ifndef __PRIORITY_QUEUE_H__
#define __PRIORITY_QUEUE_H__

#include <iostream>
#include <cassert>
#include <cstdlib>
using namespace std;

template<class Item>
class priority_queue{
    public:
        priority_queue(size_t init_cap= 5); /*NOTE decreased capacity to smaller testable
value. change back to 30*/
        ~priority_queue();
        priority_queue(const priority_queue<Item>& source);
        void operator=(const priority_queue<Item>& source);
        void pop();
        void push(const Item& entry, const int& priority);
        Item top() const;
        size_t size() const; /*{ return count; }*/
        bool empty() const;
        /* TO IMPLEMENT LATER */
        void swapPriorityQueues(priority_queue<Item>& source);
        void print();

    private:
        Item * data;
        int * priority;
        size_t first;
        size_t last;
        size_t count;
        size_t capacity;
        /* Helper Method */
        size_t next_index(size_t i){ return (i+1)%capacity;}

};


/*NOTE I AM MAKING PRIORITY TO BE INT. IT MAYBE CAN BE ITEM, BUT THAT DIDNT MAKE SENSE TO ME
IF WE USE STRINGS AS ITEMS.*/

#include "Sefa_B_HW4_Q3.cpp"
#endif
```

Implementation File: Sefa_Bujar_HW4_Q3.cpp:

```
#ifndef __PRIORITY_QUEUE_CPP__
#define __PRIORITY_QUEUE_CPP__

#include "Sefa_B_HW4_Q3.h"

template<class Item>
priority_queue<Item>::priority_queue(size_t init_cap){
    capacity = init_cap;
    data = new Item[capacity];
    priority = new int[capacity];
    count = 0;
    first = 0;
    last = 0; /*Can't seem to get this to work like queue, but works with this...*/
}


template<class Item>
priority_queue<Item>::~priority_queue(){
    delete [] data;
    delete [] priority;
    capacity = 0;
    first = 0;
    last = 0;
    /*last = -1;*/
    count = 0;
}

template<class Item>
priority_queue<Item>::priority_queue(const priority_queue<Item>& source){
    capacity = source.capacity;
    count = source.count;
    first = source.first;
    last = source.last;
    data = new Item[capacity];
    priority = new int[capacity];
    for(size_t i = 0; i < capacity; i++){
        data[i] = source.data[i];
    }
    for(size_t i =0; i < capacity; i++){
        priority[i] = source.priority[i];
    }
}
```

```
template<class Item>
void priority_queue<Item>::operator=(const priority_queue<Item>& source){
    if(this == &source)
        return;
    delete [] data;
    delete [] priority;
    capacity = source.capacity;
    data = new Item[capacity];
    priority = new int[capacity];
    first= source.first;
    last = source.last;
    count = source.count;
    size_t i = first;
    for(size_t i =0; i < capacity; i++){
        data[i] = source.data[i];
        priority[i] = source.priority[i];
    }
}

template<class Item>
bool priority_queue<Item>::empty() const{
    return count == 0;
}

template<class Item>
Item priority_queue<Item>::top() const {
    assert(!empty());
    return data[first];
}

template<class Item>
void priority_queue<Item>::push(const Item& entry, const int& prior){
    if(count >= capacity){
        capacity++;
        Item * biggerData = new Item[capacity];
        int * biggerPriority = new int[capacity];
        for(size_t i = 0; i < capacity; i++){
               biggerData[i] = data[i];
               biggerPriority[i] = priority[i];
        }
        delete [] data;
        delete [] priority;
        data = biggerData;
        priority = biggerPriority;
    }
    size_t indexPrior = last;
    bool isFound = false;
    for(size_t i = first; i < last && !isFound; i++){
        if(prior > priority[i]){
               indexPrior = i;
               isFound = true;
        }
    }
```

```
    for(size_t i = last; i > indexPrior; i--){
        data[i] = data[i-1];
        priority[i] = priority[i-1];
    }
    data[indexPrior] = entry;
    priority[indexPrior] = prior;
    last++;
    count++;
}

template<class Item>
void priority_queue<Item>::pop(){
    /*Check if queue is empty*/
    assert(!empty());
    /*THIS IS FIFO, thus, must get rid of first, NOT LAST*/
    //first = next_index(first);
    for(int i = 0; i < last-1; i++){
        data[i] = data[i+1];
        priority[i] = priority[i+1];
    }
    count--;
    last--;
}

template<class Item>
size_t priority_queue<Item>::size() const{
        return count;
}

template<class Item>
void priority_queue<Item>::print(){
    size_t index = 0;
    for(size_t i = first; i < last; i++){

        cout << "Printing entry then priority" << endl;
        cout << data[i] << " " << priority[i] <<endl;
    }
}
```

QUEUE with TWO Stacks

Header File: Sefa_Bujar_HW4_Q4.h:

```
#ifndef __QUEUE_H__
#define __QUEUE_H__

#include <iostream>
#include <cassert>
#include <cstdlib>
#include "Sefa_B_HW4_Q1.h"
using namespace std;

template<class Item>
class queue{
    public:
        queue(size_t init_cap = 5);
        ~queue();
        queue(const queue<Item>& source);
        void operator=(const queue<Item>&source);
        void push(const Item& entry);
        void pop();
        size_t size() const;
        bool empty() const;
        Item front() const;
        /*void swapQueues(queue<Item>& source);*/ /*Not required for thisquesiton. You can ust
use stack version if needed.*/
        void print();
    private:
        /*Item * data;
        size_t first;
        size_t last;
        size_t count;
        size_t capacity;*/
        stack<Item>  queueHolder;
        stack<Item>  queueTemp;
};

#include "Sefa_B_HW4_Q4.cpp"
#endif
```

10. Explanation of simulated queue: This queues works by inserting all of the values into a temporary stack, and then putting everything back in the stack that will hold the content of the queue. The reason for this is so that the first element that is inserted can be at the top of the stack (or queue). When we want to remove something, we can just pop off the top, which is the first item! Stays in FIFO order! The queue is initialized of two stacks, and all of its functions use stack helper functions to perform the main operations of push(), pop(), front(), empty(), size().

Implementation File: Sefa_Bujar_HW4_Q4.cpp:

```
#ifndef __QUEUE_CPP__
#define __QUEUE_CPP__

#include "Sefa_B_HW4_Q4.h"

template<class Item>
queue<Item>::queue(size_t init_cap){
    queueHolder = stack<Item>(init_cap);
    queueTemp = stack<Item>(init_cap);
}

template<class Item>
queue<Item>::~queue(){
    /*According to stack overflow question 677653 --> Just having this title will call stacks
deconstructor and everything will be deleted. You just need to have it. See example of class C
using class A (in this case queue calling stack.). */
    /*queueHolder.~stack();
    queueTemp.~stack();*/
}

template<class Item>
void queue<Item>::pop(){
    /*NOTE stack's verion of pop also checks assertion, but can just do it here.*/
    assert(!empty());
    /*Since stack was ordered to be in reverse, or in a way the queue would be ordered, to
remove first, just remove top!*/
    queueHolder.pop();
}

/*NOTE queueHOLDER holds the queue in the way it should, meaning that using pop/top gives it
its order.*/

template<class Item>
void queue<Item>::push(const Item& entry){
    /*Move everything from stack 1 into stack to so that order remains same*/
    while(!(queueHolder.isEmpty())){
        queueTemp.push(queueHolder.top());
        queueHolder.pop();
    }
    /*Add new entry to top of stack (where everything is reversed)*/
    queueTemp.push(entry);
    /*Re-reverse everything from stack 2 into stack 1 so that order is back.*/
    while(!queueTemp.isEmpty()){
        queueHolder.push(queueTemp.top());
        queueTemp.pop();
    }
}
```

```
template<class Item>
size_t queue<Item>::size() const{
    return queueHolder.size();
}

template<class Item>
void queue<Item>::print(){
    cout << "STACK WAY:" << endl;
    queueHolder.print();
    cout << endl << endl << "QUEUE WAY" << endl;
    while(!queueHolder.isEmpty()){
        cout << queueHolder.top() << endl;
        queueTemp.push(queueHolder.top());
        queueHolder.pop();
    }
    /*Put everything back.*/
    while(!queueTemp.isEmpty()){
        /*cout << queueHolder.top() << endl;*/
        queueHolder.push(queueTemp.top());
        queueTemp.pop();
    }
}

template<class Item>
Item queue<Item>::front() const {
    assert(!queueHolder.isEmpty());
    return queueHolder.top();
}

template<class Item>
bool queue<Item>::empty() const {
    return queueHolder.isEmpty();
}

template<class Item>
void queue<Item>::operator=(const queue<Item>& source){
    if(this == &source){
        return;
    }
    /*Since these are stacks, it will use the stack assignment operator!*/
    queueHolder=source.queueHolder;
    queueTemp = source.queueTemp;
}

template<class Item>
queue<Item>::queue(const queue<Item>& source){
    queueHolder = stack<Item>(source.queueHolder);
    queueTemp = stack<Item>(source.queueTemp);
}

#endif
```

Stack with TWO Queues

Header File: Sefa_Bujar_HW4_Q5.h:

```
#ifndef __STACK_H__
#define __STACK_H__

#include <iostream>
#include <cassert>
#include <cstdlib>
#include "Sefa_B_HW4_Q2.h"

using namespace std;

template<class Item>
class stack{
    public:
        stack(size_t init_cap = 5);
        ~stack();
        stack(const stack<Item>& source);
        void operator=(const stack<Item>& source);
        void push(const Item& entry);
        void pop();
        bool isEmpty() const;
        size_t size() const;
        Item top();
        void print();
    private:
        /*NOTE changed from pointers to standard to avoid having to write copy
constructor/assignment operator.*/
        queue<Item> stackHolder;/* * stackHolder;*/
        queue<Item> stackTemp;/* * stackTemp;*/
};

#include "Sefa_B_HW4_Q5.cpp"

#endif
```

13. Explanation of simulated stack: This stack works by inserting all of the values into a queue, which will hold everything in FIFO order. When something needs to be popped off, I knew I needed to make sure it was in LIFO order. So what I do is, I push everything from the queue that holds the stack content into a temporary queue, all but the last item. The last item is then popped off and all other items are returned to that queue. The same applied for getting the top, the function pushes all but the last into a temporary queue, then returns the last item, then pushes everything back onto the original queue. Stays in LIFO order! The stack is initialized of two queues, and all of its functions use queue helper functions to perform the main operations of push(), pop(), top(), empty(), size().

Implementation File: Sefa_Bujar_HW4_Q5.cpp:

```
#ifndef __STACK_CPP__
#define __STACK_CPP__

#include "Sefa_B_HW4_Q5.h"

template<class Item>
stack<Item>::stack(size_t init_cap){
    stackHolder = queue<Item>(init_cap);
    stackTemp = queue<Item>(init_cap);
}

template<class Item>
stack<Item>::~stack(){

}

template<class Item>
stack<Item>::stack(const stack<Item>& source){
    stackHolder = queue<Item>(source.stackHolder);
    stackTemp = queue<Item>(source.stackTemp);
}

template<class Item>
void stack<Item>::operator=(const stack<Item>& source){
    /*Check if its the same thing*/
    if(this == &source){
        return;
    }
    /*CALL the assignment operator from stack which handles everything it needs to.*/
    stackHolder = source.stackHolder;
    stackTemp = source.stackTemp;


}

template<class Item>
bool stack<Item>::isEmpty() const{
    /*Call the queue's empty function.*/
    return stackHolder.empty();
}

template<class Item>
size_t stack<Item>::size() const{
    return stackHolder.size();
}

template<class Item>
Item stack<Item>::top(){
    /*Let's make sure we aren't empty.*/
    assert(!stackHolder.empty());
    Item ret = Item(); /*Return the default if nothing.*/
    /*Move everything but last element.*/
```

```
    while(stackHolder.size()!=1){/*NOTE check size is up to 1 because want to return last
one.*/
        stackTemp.push(stackHolder.front());
        stackHolder.pop();
    }
    /*Store front "last item" to return*/
    ret = stackHolder.front();
    /*Get rid of last item*/
    stackHolder.pop();
    /*Put everything back into the queue that holds the stack.*/
    while(!stackTemp.empty()){
        stackHolder.push(stackTemp.front());
        stackTemp.pop();
    }
    /* NOTE NOTE NOTE PLACE BACK THE ORIGINAL AT THE END.*/
    stackHolder.push(ret);
    return ret;

}

template<class Item>
void stack<Item>::push(const Item& entry){
    /*NOTE NOTE NOTE everything just linerally pushes into queue called stack Holder and then
popping wll handle getting rid of the last element.*/
    stackHolder.push(entry); /*Add new to holder*/
}

template<class Item>
void stack<Item>::pop(){
    /*Let's make sure we aren't empty.*/
    assert(!stackHolder.empty());
    /*Move everything but last element.*/
    while(stackHolder.size()!=1){/*NOTE check size is up to 1 because want to return last
one.*/
        stackTemp.push(stackHolder.front());
        stackHolder.pop();
    }
    /*Get rid of last item*/
    stackHolder.pop();
    /*Put everything back into the queue that holds the stack.*/
    while(!stackTemp.empty()){
        stackHolder.push(stackTemp.front());
        stackTemp.pop();
    }
}

template<class Item>
void stack<Item>::print(){
    /*Call queues print function.*/
    stackHolder.print();
}

#endif
```